Analysis of a Disk Operating System (プログラムの基礎理論)

AUTHOR(S):

OKUI, JUN; TOKURA, NOBUKI; KASAMI, TADAO

CITATION:

OKUI, JUN ...[et al]. Analysis of a Disk Operating System (プログラムの基礎理論). 数理解析研究所講究録 1973, 189: 101-117

ISSUE DATE:

1973-10

URL:

http://hdl.handle.net/2433/107217

数理解析研究所講究録
189巻 1973年 101-117

ANALYSIS OF A DISK OPERATING SYSTEM

Jun OKUI, Nobuki TOKURA and Tadao KASAMI

Faculty of Engineering Science, Osaka University

1. Introduction

A PDP-11 dual system with an interprocessor communications channel DA11-D installed at Department of Information & Computer Sciences, Osaka University in February 1972. A disk operating system (DOS) is one of manufacturer-supplied systems. Our group started analysis of the DOS monitor program. The motivation for this is the following. First, it is one of our projects to develop a DOS for the dual system and one of our approach was to modify the existing DOS. Second, the analysis may be interesting theoretically. Equivalence of program schemata is in general undecidable. Hence, one of the next important problem may be to specify classes of program schemata of which equivalence is decidable. In this direction, some model for assemblers and others were described in [1]. There are two reasons for us to study systems programs mainly. First, the classes of system programs are "non-numeric" programs and equivalences in the class are expected to be more easily decidable. It is shown that the equivalence of programs of numerical computations is undecidable even if they are of very simple structures [12]. Second, it is a real problem to provide some systematic tools for developing programs of this type.

In this paper, some observations on PDP-11 DOS monitor program are shown. A model of some parts of the DOS monitor is described and related subjects are discussed.

2. The DOS monitor program

2.1 Loop structure

The analysis of the DOS monitor has been done by reading the listing and by using tracers. The DOS monitor program consists of many modules and the following exposition is mainly on module's level.

The DOS monitor looks very intricate and difficult to understand partly because it is written with various programming techniques and partly because the documentation is not complete. However, it is observed that the DOS monitor is rather simple with respect to structure.

It is a problem to select a suitable measure of complexity of programs. The prime factor seems to be the complexity of loops. The factors of loop complexity are 1)existence of overlaps, 2)depth of nesting, 3) number of entry points and the number and the destination of exit points, though these factors are not independent.

In the DOS monitor, the loop structure is very simple. Most loops have one entry and are properly nested. The depth of nest is in most cases less than 3.

A number of loops found in the monitor are listed.

1. loops for wait
2. loops for a fixed number of right (or left) shifts
3. loops for calculating the parity of a word
4. loops for block transfer
5. loops for searching and updating a table or a bit map
6. loops for traversing a linked list
7. loops for searching and updating a file directory
8. loops for inserting into a priority queue

The uses of loop construction may be classified as follows:

a) loops for wait——1

b) loops deal with dynamic data structure——4. 5. 6. 7. 8.

c) loops in place of "straight-line" code——2. 3. 4.

2.2 Folding

Even though the loop structure is simple, the DOS monitor's structure looks still complicated. The prime reason of this is considered as a reflex of one of the design goals to minimize the memory space for the monitor.

In order to shorten the code, large common parts are treated as subroutines or modules and in each module, common parts are shared as much as possible. Fig.1 shows the control flow of a module in unfolded form. In this example, the routine is 67 words in length and the unfolded one is 212 words in length. The "folding coefficient" is about $212/67 \fallingdotseq 3.2$. The folding coefficient which is calculated for some modules ranges from 1 to 360. This difference is caused by "folding" the code so that the common parts are shared by many control paths.

The "folding coefficient" is a measure with the tree equivalent as a basis. The reason to use the tree-equivalent as a basis is that the tree is in a sense the simplest structure and it is easy to obtain the tree equivalent for a given program. There may be another choice of a basis. To measure the control-flow irregularity, it seems to be better to use a well-formed program equivalent with respect to the control flow. A definition of a well-formed program is given in [11]. It is found that all the modules can be written as well-formed flowchart in the sense of [11] with no extra cost such as node-splitting.

The flowchart of Fig.1 can not be written using only IF statements apart from the loop. If the use of GOTO statements is banned, then "node splitting", that is, duplications of some parts of codes is inevitable.

In this case, the length of equivalent program becomes 91. The ratio 91/67 ≒ 1.4 seems to reflect the complexity with respect to GOTO statements. Dijkstra warned of the control-flow irregularity induced by the free use of GOTO statements.[2] In real operating systems, (conditional and unconditional) branches are used at the maximum.

2.3 Data structures

Various data structures are used in the DOS monitor.

1. *Stack* The PDP-11 has stack processing capabilities. The stack pointer (general register 6) maintains a stack for the nested handling of interrupts and subroutine calls. All of the general registers can maintain stacks under program control.
2. *Table* There are a number of system tables of fixed sizes. The bit maps are linked tables. There is a dynamic assign table (D.A.T.) whose size changes dynamically. For each data set assign command, a new entry whose size is not fixed is linked to the D.A.T.
3. *File directory* The file directory consists of a master file directory (MFD) and a number of user file directories (UFD) (Fig.2). A UFD is a linear list of fixed size blocks each of which contains a file name and associated information. The MFD is a linear list of fixed size blocks each of which contains a user identification code, a pointer to the user's UFD and other information. Each file-structured device has its file directory in it.
4. *Queue* The DOS monitor uses "priority" queues. The queue is formed for each device and is a linear list of dataset data blocks (DDB) which represent I/O requests. If an I/O is requested for a device A, then a DDB with parameters of the I/O request is inserted into the queue for A on the basis of high-priority first and first-in first-service within the same priority class. If the current I/O service is completed, then the

top DDB is deleted and the next DDB is serviced.

5. *Linear list* If a dataset is initialized, a DDB is created and the DDB is linked to the initiated DDB chain. If a dataset is released, the DDB is removed from the chain.

These data structure are not independent. A DDB belongs both to the initiated DDB chain and to the priority queue by using two pointers. There are the other pointers in DDB which point other data structures and there are pointers which point to DDBs. Thus, the whole data structure in the DOS monitor is rather complicated. But, the routine which handles the initiated DDB chain is different from the one which handles the priority queue and only one of these routines is active at any time. It can be said that each routine treats a substructure which may be regarded as a linear list.

3. A model of the DOS monitor modules

The models of Ianov, Paterson and others might be considered rather as models of numerical computations[7]. They have a definite number of registers. The loops in them permit computations of arbitrary depth. On the other hand, in the DOS monitor, there are no loops of the type such that the depth of computation increases unboundedly. It may be natural to consider the DOS monitor have an arbitrary number of data cells or registers. The study on the program schemata with data structures is expected.

In this paper, a simple model is shown. Some other works are reported in [4] [5].

3.1 Linear list operations

The functions of loops for data structures amount to the operations on linear lists. A linear list is a set of $n \geq 0$ blocks $B_1, \ldots, B_n$. Each block has a number of fields and one of the fields contains a pointer to the next block or NIL.

The following operations are found in the DOS monitor.

a) Insert a new block just before a specified block.

b) Delete a specified block.

c) Change the contents of the fields of a specified block.

d) Search the list for the first occurence of a block which satisfies a condition.

Let us introduce three macros.

1) Insert (ℓ, b, y) (Fig. 3a)

ℓ specifies the block before which a new block pointed by the pointer b is to be inserted and the pointer to the old block is stored in the field y of the new block.

2) Delete (ℓ, y) (Fig. 3b)

3) Search (ℓ, y, P, L)

ℓ points to the start block B_1 of the list searched for. For $i = 1, 2, \ldots$, the pointer in the field y of block B_i points to the next block B_{i+1}. P is a predicate whose truth value can be determined only by the parameters given before the search is executed and the contents of fields of a block. When the search is completed, L will contain the pointer to the first block which satisfies the predicate P and L will contain NIL if there are no blocks in the list which satisfy P.

The loops which concern data structures amount to the implementations of this search macro.

For example, the insertion of a new DDB B with priority p into a priority queue is written as follows:

Search (Priority queue, link, p > contents of the priority field, L)

Insert ($\overline{L}$, pointer to B, link)

where $\overline{L}$ is the contents of variable L.

Each routine with neither loops for wait nor some exceptional loops can be written as a loop-free program by using these macro instructions.

3.2 D machines and E machines

As a model of operations on linear lists, the LF machine model was introduced [3]. In this paper, some other extended machine models are shown. First, consider a D machine. D machine M has an input tape and an output tape. Each tape is divided into fields whose sizes are not bounded. This covers the cases of an arbitrary number in the binary notation or symbol strings with arbitrary length. M has instructions such as $SCAN_R$, $SCAN_L$, OUTPUT and STOP. Its behaviour may be well described by a flowchart. Each branch point of the flowchart is labelled with a predicate of the form $P_i(R, a_{i1}, \ldots, a_{in_i})$ where R is a register of M and $a_1, \ldots, a_n$ are parameters given before M starts its action. (These parameters may not be bounded.) If a $SCAN_R$ (or $SCAN_L$) is executed, the contents of the field under the input head is transferred to the register R and M moves its input head one field to the right (or to the left, respectively.) The OUTPUT has three types of operands; R, $f_j(R, a_{j1}, \ldots, a_{jm_j})$ and a_j. If OUTPUT X is executed, the contents or value of X is written out on the output tape as a field and M moves its output head to the right.

M is a kind of two-way automaton with output. It may be in a loop and will never reach the end of the input tape and it may output on the same field many times. M is said to be pertinent if the following conditions are fulfilled.

1) M is never in a loop.

2) M never rescans the field on the input tape whose contents are transferred to R and then OUTPUT R or OUTPUT $f_j(R, a_{j1}, \ldots, a_{jm_j})$ are executed.

P0: It is effectively decidable whether a D machine is pertinent.

The proofs of P0 through P4 are omitted.

This D machine can model the operations on rather broad class of data structures; such as linear lists, file directories, structures in PL/1. For example, let us consider a file directory (Fig.2). The file directory is transcribed on the input tape so that the input tape looks as # M_1, U_{11}, U_{12}, ..., U_{1n_1} : M_2, U_{21}, ..., U_{2n_2} : ... : M_m, U_{m1}, ..., U_{mn_m} # where #, : are delimiters, U_{i1}, ..., U_{in_i} is a UFD for the user i and $\{M_1, \ldots, M_m\}$ is a MFD. U_{ij}'s and M_i's are blocks whose fields are placed on the tape in order. M can simulate various operations on the file directory. For example, DELETE operation which searches a file name given as a parameter and deletes the block with the file name is simulated as follows. M searches the field with the file name block by block and if a block contains the file name, then M outputs nothing (i.e. delete) and if a block under scan does not contain the file name, M returns its head to the start of the block and copies the block on the output tape.

The equivalence of two D machines M_1 and M_2 are defined as follows:

M_1 and M_2 are equivalent if given the same input tape and the same parameters, M_1 and M_2 produce the same output tapes starting from the initial state with the same contents of R, where the "same" output tapes mean the strong equivalence in the sense that they are the same under any interpretations.

Pl: It is decidable whether pertinent D machines M_1 and M_2 are equivalent.

Outline of the proof: Each field on the input tape is classified according to values of all the predicates in M_1 and M_2. If there are k different predicates, then each field is classified into 2^k classes. A new sets of 2^k symbols $c_1, \ldots, c_{2^k}$ is used and each field is replaced by a symbol. Each field on the output tape is replaced by a letter R or a symbol string $f_j(R, a_{j1}, \ldots, a_{jn_j})$ or string a_j according to the contents of it. In this way, M can be replaced by a two-way finite automaton with output. This can be reduced to

a sequential machine. Hence, the equivalence is decidable.

The D model is rather powerful but it does not admit to rescan rewritten parts. To avoid this restriction, a modified version of the D machine (E machine) is considered. An E machine M has no output tape but M has a tape whose initial contents is regarded as an input to the machine and M can rewrite any fields and rescan them. The restrictions to be pertinent are no more necessary. If M is permitted to scan the tape indefinitely, then M can simulate a Turing machine. Thus, it is required that M scans each field less than K times for a fixed K. This restriction is necessary also from technical reason because each field must be classified according to the values of predicates whose operands may be a composite-function's value. On the other hand, this restriction is not severe for the modelling of the DOS monitor program because there can not be found a module which operates on a data structure indefinitely. Then, we have

P2: The equivalence of scan-limited E machines is decidable.

Let us consider the power of the D machines and E machines. Let T_k be the class of the input tape t's such that (1) $t=S_0$ (2) S_i is of the form $\#_i(FS_{i+1})^*F\#_i$ for i=0,..., k-1 and (3) S_k is of the form $\#_kF^*\#_k$, where F stands for a field or an empty field. T_k is a nested tape of depth $\leq k$. The subtape S_i's are of level i.

The D machines and E machines can do the following operations on the tapes of T_k.

1) To modify fields in each level.
2) To delete fields or a subtape S_i.
3) To insert fields or a subtape S_i.
4) To change the level of a subtape S_i by rewriting the delimiters in S_i.

The followings are some examples of data structures [8] such that their structure can be represented as a tape of T_k for some k and some operations

on them can be simulated by the above operations.

a) linear list, circular list, doubly linked list

b) a class of trees, e.g. COBOL structures with bounded level but an unbounded number of items on each level.

c) a class of graphs, e.g. graph structures with bounded level[9].

The correspondence is straightforward. The D and E machines can be regarded as machines which traverse and change the structure dynamically. The amount of the change is bounded, though the model might be well as a model of the DOS monitor modules. The method of traversing on the data structure is restricted in a sense, indeed, the equivalence becomes undecidable if it is admitted to traverse on a data structure freely. Now, consider a machine called an F machine. The F machine M scans a possibly infinite 2 dimensional array A whose elements are A[i, j] ($i \geq 0$, $j \geq 0$). M only moves its head on A and does not change the array. M can only detect that it is on the boundaries A[o,j] and A[i, 0] ($i, j \geq 0$). When it halts, it answers yes or no. The F machine is very simple as compared with the D and E machines. However, we have:

P3: The equivalence of the F machines is not decidable in general.

This is because the F machines can simulate 2-counter machines directly.

The D and E machines can simulate operations on a class of trees. We have a stronger result. Let us consider a class of G machines. G machine M has a head and M moves its head on a rooted tree. Each node of the tree is considered as a block with a number of fields which are not necessarily bounded. M's head moves along the edges to either direction so that it is an extension of a two-way automaton, where the linear input tape is replaced with a tree. M starts in its initial state with its head scanning the root. If M moves off the tree, M halts. A tree will be accepted by M if and only if M eventually moves off the root at the same time it enters

a final state. G machines M_1 and M_2 are said to be equivalent if they accept the same set of trees. We have,

P4: The equivalence of the G machines is decidable.

We have some extended results and these will be appeared in the following papers.

4. An example of the DOS monitor specification

In the preceeding sections, the DOS monitor is analyzed and a model is constructed. This is "bottom up" approach. The process to develop a program is "top down" approach. Some systematic and efficient methods to develop programs are wanted. The starting point is to describe the program specification precisely and independently of implementation methods. The next step is to implement, though its implementation itself may be carried out step by step.

An example of the user-level specification for a part of file processing in the DOS monitor is shown in Fig.4. It describes the correct or permissible monitor request macro sequences for a certain dataset and the responses of the DOS monitor.

A file is conceptually regarded as a byte string which is divided into lines by line delimiters. There are several modes of .READ/.WRITE services, but for simplicity only Formatted ASCII mode is described. .READ/.WRITE macros transfer one line at each execution from a file to the user's line buffer and vice versa. It can be conceptually considered that there is a pointer associated with each file which points to the start byte position of the next line transfer. The parameters used are as follows:

a: dataset name

b: device name

c: file name

d: user's line buffer address

e: error return address

The predicates are as follows:

P_2: Error return address e is specified in the macro.

$P_3(a)$: No device assigned to the dataset a.

$P_4(a)$: The device assigned to a is not an input device.

$P_5(d,a)$: Transfer of one line is completed from a to the user line buffer whose address is d.

$P_6 = P_{61} \vee P_{62} \vee P_{63} \vee P_{64}$, where $\vee$ denotes "OR".

$P_{61}(a)$: The device assigned to a is not a file-structured device and is already opened.

$P_{62}(c)$: There is no file c or c is already opened for output.

$P_{63}(c)$: c is opened too many times.

$P_{64}(c)$: An attempt was made to access a file which the protection code prohibits.

$F_1(d,a)$: A line pointed by the pointer of the dataset a is transferred into the user line buffer d. The pointer moves to the next line. If the line is longer than the size of the line buffer, then an error code is set on the communication block of the line buffer. The other error conditions are similarly indicated on the error code .

All that have meanings for users are specified but nothing more. It is independent of how the monitor is implemented. Indeed, users are not concerned about how the predicate P_3 is evaluated. It is enough to describe that if P_3 is true then the monitor prints a message to the user that the dataset is not yet assigned. Apart from the discussion whether this description is easier to understand than usual user's manual, it seems to cover ambiguities and lack of dynamic description which are usual with manuals.

From this user-level specification, the specification for the implementer is derived in the following way. The above specification is concerned with distinct datasets. In general, a number of datasets are used at the same time. Then, a kind of product of the specification diagram is considered. There are predicates whose value can be determined in connection with other datasets, e.g. P_{62}. The necessary information to determine the value of the predicates is selected and collected in a suitable form. Thus, for example the notion of a file directory will be introduced naturally. By collecting responses for a monitor request macro, a specification of a module which processes the monitor request macro is obtained. The specification is detailed step by step by specifying the suitable data structures and their format and by subdividing the program into independent modules. At each step, there are many possible choices. It is a problem to be solved how to make good choices at each step.

One of purpose for us to refer the specification in this paper is the following. It may be interesting to compare the observations and the some detailed specifications on the implementer level derived from the user level specification. Some thoughts seem to justify the simple loop structure of the DOS monitor. If there are found some degree of coincidence, the observations stated might be considered not a characteristic of the PDP-11 DOS monitor only but a characteristic common to many operating systems in a extent.

The authors are grateful for the discussions by the members of the Kasami Laboratory.

References

1. S.Hatanaka, K.Taniguchi, M.Fujii, N.Tokura and T.Kasami, "Equivalence problems of a certain class of program schemata," Papers of Technical Group on Automata and Information Theory, I.E.C.E., Japan (Jan. 1970).

2. E.W.Dijkstra, "GOTO statement considered harmful," CACM,11, 147-148 (1968).

3. J.Okui, N.Tokura and T.Kasami, "Analysis of a control program ___ a model of linear list processing machine," Papers of Technical Group on Automata and language, I.E.C.E., Japan (April, 1972).

4. T.Hosomi, N.Tokura and T.Kasami, "Some program schemata for file processing," Op. cit. (June, 1972) The full paper is in preparation.

5. S.Furuta, N.Tokura and T.Kasami, "Program schemata with pushdown store," Op. cit. (Oct. 1972).

6. J.Okui, N.Tokura and T.Kasami, "A specification method for systems programs," Joint Convention Records of Electrical Engrs. in Kansai 1972.

7. D.C.Luckham, D.M.R.Park and M.S.Paterson, "On formalized computer programs," JCSS 4, 220-249 (1970).

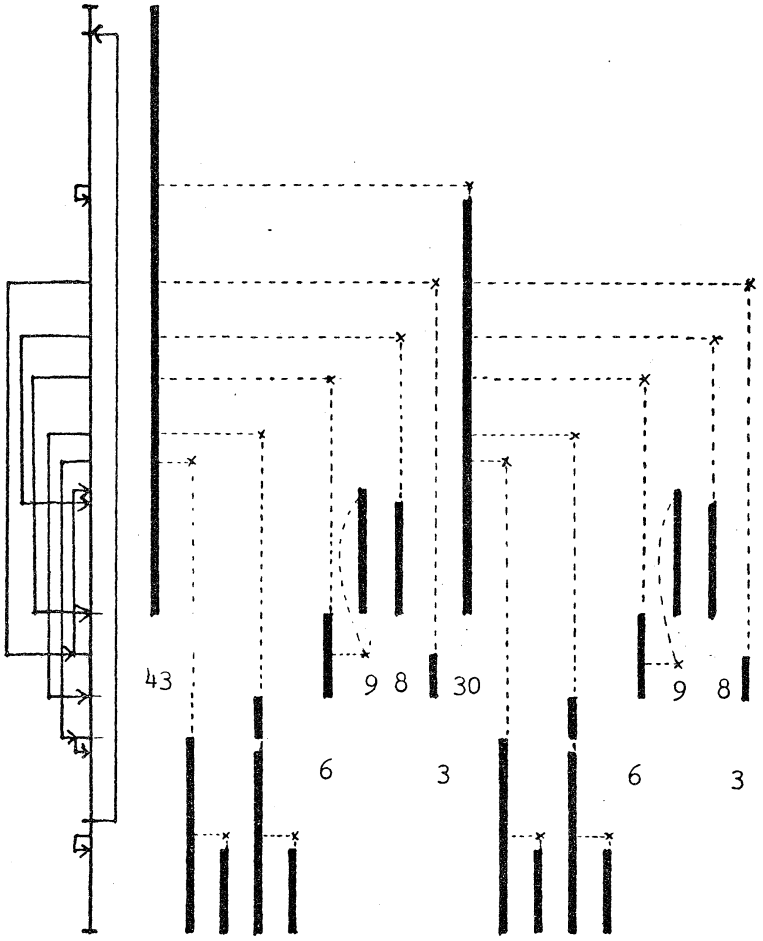
8. D.E.Knuth, "The Art of Computer Programming, vol.1," Addison-Wesley, Massachusetts, (1968).

9. J.L.Pfaltz, "Graph Structures," JACM,19, 411-422 (1972).

10. PDP-11 Disk Operating System Monitor, Programmar's handbook, DEC-11-MWDA-D and Listing, LBKIT-11-DSLS-01.

11. W.W.Peterson, T.Kasami and N.Tokura, "On the capabilities of WHILE, REPEAT and EXIT Statements," to appear in C.ACM.

12. Tadao Kasami and Nobuki Tokura, "Equivalence problem on programs without loops," Trans. IECE, Japan 54-C 657-658 (1971).



67 14 6 16 6 14 6 16 6

Original Unfolded (212/67≒3.2)

Fig. 1 Control Flow of a Module
(The loop is not taken into account.)

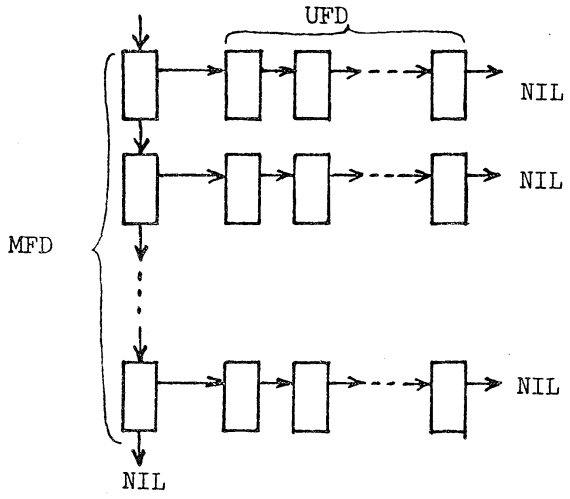
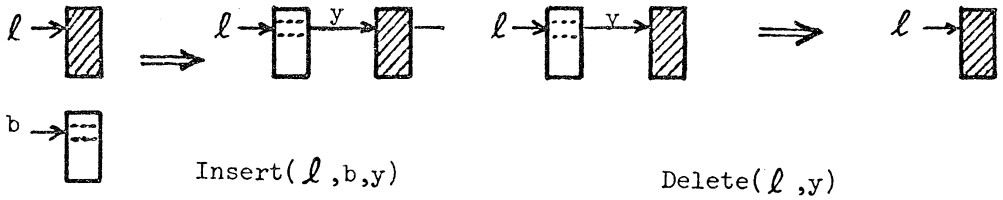


Fig.2 File Directory



Insert(ℓ ,b,y)
a)

Delete(ℓ ,y)
b)

Fig.3 Insert and Delete

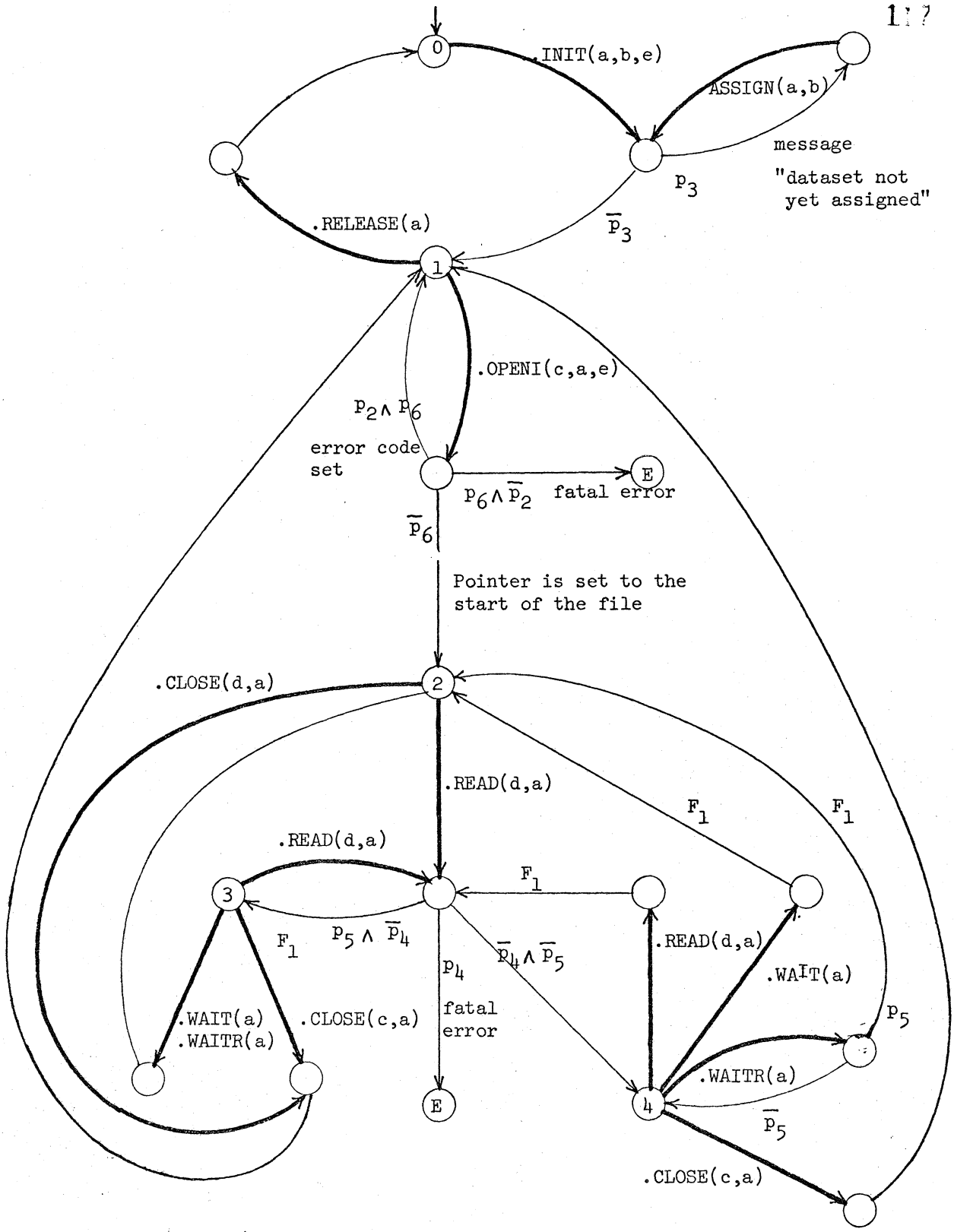


Fig.4 An example of the functional specification
(The bold lines represent the user's action.)